Étude et développement d'un module de contrôle pour une plate-forme de simulation numérique

Matthieu Haefele, David Vigier, Eric Violard, Florence Zara

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

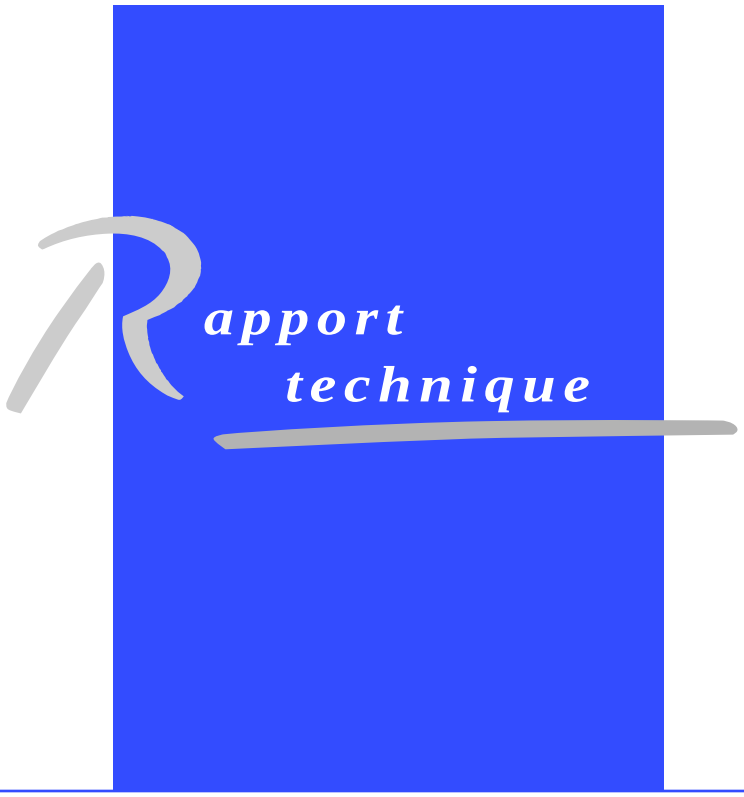
Étude et développement d'un module de contrôle pour une plate-forme de simulation numérique

Matthieu Haefele — David Vigier — Eric Violard — Florence Zara

N° 0299

Septembre 2004

Thème NUM D



ISSN 0249-0803

Étude et développement d'un module de contrôle pour une plate-forme de simulation numérique

Matthieu Haefele [*] [†], David Vigier [*], Eric Violard [‡], Florence Zara [*]

Thème NUM D —Modélisation, simulation et analyse numérique
Projet CALVI

Rapport technique n° 0299 —Septembre 2004 — 56 pages

Résumé : Ce travail a comme objectif d'étudier diverses solutions pour la mise en place d'une plate-forme de simulation numérique. Celle-ci doit pouvoir rassembler plusieurs programmes développés au sein du projet INRIA-CALVI, dont le but consiste en l'étude mathématique et numérique et la visualisation de divers problèmes issus essentiellement de la physique des plasmas et des faisceaux de particules.

Ce rapport technique présente le développement d'un module de contrôle et propose une API (Application Programming Interface) à laquelle doivent se conformer les programmes destinés à tourner sur cette plate-forme. Il présente également le langage de script nommé python, ainsi que l'utilisation d'outils permettant d'étendre ses possibilités par des langages compilés.

Mots-clés : plate-forme, simulation numérique, codes numériques, physique des plasmas, langage python

[*] LSIIT-IGG, UMR CNRS-ULP 7005

[†] Financement de thèse "Région Alsace", "Institut de Recherche Mathématique Avancée (IRMA) de Strasbourg" et "Action de Recherche Coopérative (ARC) PLASMA"

[‡] LSIIT-ICPS, UMR CNRS-ULP 7005

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 —Télécopie : +33 3 83 27 83 19

Development and Studie of a Control Module for a Platform of Numerical Simulation

Abstract: This work dealt with setting up a platform of numerical simulation. It must be able to gather all the programs developed in the INRIA-CALVI project, a project whose goal is the mathematical and numerical study and the visualization of different problems from plasma and beam physics.

This technical report presents the development of a control unit and proposes an API (Application Programming Interface) that all programs, which are to run on this plateform, should conform to. Then it presents the python script language, and also the use of tools that allow to extend its possibilities with compiled languages

Key-words: platform, numerical simulation, numerical programs, physic of plasmas, python language

1 Introduction

Le projet INRIA-CALVI (CALcul et VIsualisation) est commun à l'Institut Elie Cartan (UMR 7502, INRIA, CNRS et Université Henri Poincaré, Nancy), à l'Institut de Recherche Mathématique Avancée (UMR 7501, CNRS et Université Louis Pasteur, Strasbourg) et au Laboratoire des Sciences de l'Image, de l'Informatique et de la Télédétection (UMR 7005, CNRS et Université Louis Pasteur, Strasbourg) avec une collaboration étroite avec le Laboratoire de Physique des Milieux Ionisées (UMR 7040, CNRS et Université Henri Poincaré, Nancy).

Ce projet est consacré à l'étude mathématique et numérique et à la visualisation de divers problèmes issus essentiellement de la physique des plasmas et des faisceaux de particules. C'est pourquoi ces différentes équipes de recherche représentent les différents domaines scientifiques à maîtriser pour parvenir à progresser dans la compréhension des plasmas. Les physiciens de l'IECN effectuent des expériences numériques en recourant à la modélisation des phénomènes ainsi qu'à leur résolution en utilisant la simulation numérique. Les mathématiciens de l'IRMA et de l'UHP mettent au point des nouvelles méthodes numériques dans le but d'améliorer le temps d'exécution et la précision de ces simulations. Enfin, les informaticiens du LSIIT travaillent sur la parallélisation des codes de simulations ainsi que sur la visualisation des résultats issus de ces codes.

Il est à noter qu'un modèle couramment utilisé pour étudier le comportement des particules du plasma se base sur l'équation de Vlasov couplée avec les équations de Maxwell ou Poisson pour décrire les champs électriques/magnétiques [3, 2, 6, 4]. Le but de la simulation est alors de calculer la fonction $f(t, x, v)$ caractérisant pour chaque espèce de particules la distribution des particules pour un instant t donné, pour un vecteur position x et un vecteur vitesse v dans l'espace des phases pour chaque espèce de particules. L'espace des phases étant un espace permettant de représenter l'évolution d'un système d'une manière plus exploitable que l'espace Euclidien classique, il est couramment utilisé dans de nombreux domaines (en mécanique quantique ou dans l'étude de phénomènes chaotiques) et donc particulièrement dans le cadre de l'étude de la cinétique des plasmas. Cet espace est simplement le produit de l'espace ordinaire par l'espace des vitesses. En d'autres termes, un point matériel est repéré dans cet espace par les coordonnées (x, y, z) de son vecteur position x ainsi que par celles de son vecteur vitesse v, notées (vx, vy, vz). Il existe deux principales approches pour la résolution numérique de cette équation : la première se base sur un calcul par particules "Particle-In-Cell" (PIC) et la seconde consiste à résoudre l'équation de Vlasov sur un maillage de l'espace des phases.

La technique de résolution numérique de Vlasov par code PIC approxime le plasma par un ensemble fini de particules. Ce type de méthode va donc étudier le comportement du plasma en calculant l'évolution de la position et de la vitesse de chacune des particules du système. La trajectoire d'une particule étant calculée grâce aux courbes caractéristiques de l'équation de Vlasov alors que les champs sont calculés sur une grille représentant l'espace des phases. L'avantage de ces méthodes résident dans le fait que l'on obtient des résultats satisfaisants pour un nombre relativement faible de particules. Cependant, l'inconvénient majeur inhérent à ce type de méthodes se situe dans une quantité importante de bruits numériques qui peut être trop conséquente pour obtenir une précision suffisante de la fonction de distribution. De plus, l'augmentation du nombre de particules nécessaire pour accroître la précision n'est pas performante. En effet, le bruit numérique ne décroît que d'un ordre de $1/\sqrt{N}$ avec N le nombre de particules du système.

Pour pallier la faible précision des codes PIC, une méthode de discrétisation de l'équation de Vlasov utilisant un maillage de l'espace des phases a été proposée [5, 1]. Ces techniques ont l'avantage d'obtenir une précision importante. Cependant en général, elles sont généralement plus gourmandes en calcul que les méthodes PIC.

Un des objectifs du projet INRIA-CALVI réside dans le développement d'une plate-forme de simulation numérique qui permettra de capitaliser au sein d'un même environnement, l'ensemble des recherches menées dans les différents laboratoires ces dernières années. Le sujet de ce travail consiste donc à étudier les différentes manières d'implanter cette plate-forme, tant au niveau du choix des outils à utiliser que de l'utilisation et la documentation de ces outils.

En effet, au jour d'aujourd'hui, c'est-à-dire en mai 2004, il existe 7 codes de simulation distincts. Tous ces codes résolvent l'équation de Vlasov, mais ont chacun des particularités soit au niveau de la modélisation et la conservation de certaines grandeurs physiques, soit au niveau de la méthode numérique utilisée. Un tableau récapitulatif de ces différents codes figure en annexe B (page 38). Mais le point commun à l'ensemble de ces codes réside dans leur décomposition en deux parties. Une première partie résout l'équation de Vlasov en partant d'une distribution de particules et d'un champ électrostatique/-électromagnétique. Puis l'autre partie calcule ce champ grâce à la distribution de particules.

Quelque soit le code, ces deux concepts sont donc présents et clairement identifiables. L'idée de la plate-forme est alors de "couper" tous ces codes en deux solveurs distincts : un "solveur Vlasov" et un "solveur Champ", et de les faire collaborer au sein d'une troisième application que sera la plate-forme. Ainsi tous ces solveurs seront interchangeables, et les cas-tests physiques possibles, c'est-à-dire les tests permettant leur validation d'un point de vue physique, seront d'autant plus nombreux. De plus, lors de la mise au point d'un nouveau solveur, l'utilisation des solveurs existants imposera au développeur un cadre qui évitera de nombreux bugs et un temps de développement/debug moins important.

Le problème réside donc dans l'élaboration d'une telle plate-forme et donc dans la réalisation du découpage et la liaison des sept codes qui sont actuellement compilables et exécutables séparément. Pour cela nous allons partir d'un code semi-langrangien adaptatif nommé **Obiwan** en vue de son intégration à la plate-forme.

Les meilleures solutions ont du être recherchées pour résoudre ce problème et étudier les outils appropriés pour être en mesure de les utiliser dans la phase de développement. Le langage de programmation python a été choisi pour être au coeur de cette plate-forme, et des programmes `swig` et `f2py` pour la création d'extension à python. Nous verrons donc quelles ont été les solutions et laquelle nous avons retenue. Enfin, nous entrerons dans le vif du sujet, à savoir la partie développement, qui comporte une description de l'API (l'Interface de Programmation ou Application Programming Interface) que nous avons adopté puis l'implantation proprement dite.

2 Etude du problème

Dans cette section nous allons, dans un premier temps, analyser le problème. Puis nous verrons quelles sont les solutions techniques à notre disposition. Enfin, nous expliquerons les raisons de notre choix.

2.1 Position du problème

Afin d'entamer la construction de la plate-forme, et plus particulièrement, de développer le module de contrôle au coeur de celle-ci, il convient de faire le point sur les divers composants qui constitueront les modules de calcul et visualisation.

Nous disposons dans ce projet de plusieurs programmes déjà fonctionnels et réalisant, à partir d'une distribution de particules initiale et d'un champs électrique extérieur fixé,

- le calcul de la densité de particules par la résolution de l'équation de VLASOV grâce à une méthode Semi-Lagrangienne ;
- le calcul du champs de force résultant de cette distribution basée sur les équations de MAXWELL ou de POISSON ;
- la visualisation de l'évolution du faisceau de particules.

Sans s'étendre sur la physique très présente dans ce projet, on peut constater que les divers *solvers* (vlasov-maxwell, vlasov-poisson) partagent des données qu'il faudra normaliser. De plus, ils travaillent en interaction dans une boucle, illustrée par la FIG. 1, que le module de contrôle devra recréer. Cette boucle correspond à un pas de temps dans la résolution numérique.

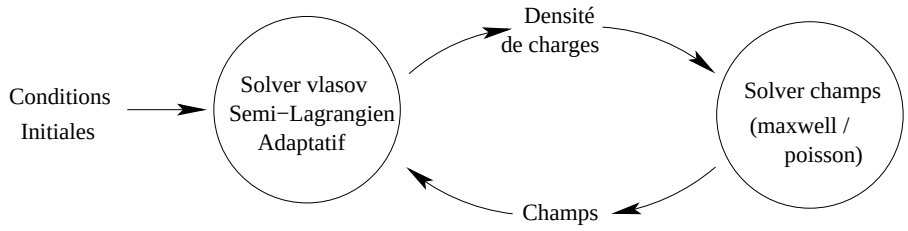


FIG. 1 – Boucle d'interaction

On souhaiterait donc que ces différentes méthodes de résolution numérique puissent être combinées de manière simple. L'un des problèmes à résoudre est donc l'unification des interfaces des *solvers*. Pour cela, il faudra proposer une interface de programmation (API) commune.

Ensuite, les programmes dont nous disposons sont écrits dans différents langages de programmation. Il s'agit de ceux donnés en annexe B. Le problème qui se pose alors réside dans le fait que ces langages n'ont pas une politique commune de gestion de la mémoire et des données. Par exemple, les tableaux en fortran sont représentés par une suite de colonnes alors qu'en C++ ils sont représentés par une suite de lignes. Le module de contrôle doit donc être capable de traduire et transmettre les données d'une partie à l'autre de la plate-forme.

Il est à noter que ce travail s'est largement basé sur le code **Obiwan** en C++ et donc l'implantation actuelle de la plate-forme en dépend fortement alors qu'elle devrait être plus générale. Ce point fait par-

tie des perspectives décrites plus loin dans le rapport.

Face à ces contraintes, de nombreuses solutions, plus ou moins adéquates, se proposent à nous. Nous allons donc tâcher de choisir la plus appropriée.

2.2 Solutions et choix

La première solution venant à l'esprit est de construire un unique exécutable contrôlé par une fonction *main* en C par exemple et contenant l'ensemble des programmes cités. L'avantage principale de cette méthode est l'efficacité. En effet, une telle plate-forme, implantée en langage bas-niveau, s'exécute rapidement tant au niveau des calculs que de celui de la traduction des données. En revanche, l'exécutable en question demande la quantité de mémoire nécessaire à tous les *solvers* simultanément. De plus, la modification d'une partie du programme nécéssite de le recompiler intégralement.

Une autre solution consiste à utiliser une couche logicielle intermédiaire pour mettre en relation les différents codes. Python s'impose alors naturellement à nous comme le langage le mieux adapté. Il s'agit d'un langage de script orienté objet possédant de nombreuses qualités. Il a fait ses preuves dans beaucoup de projets scientifiques. Pour n'en citer qu'un, le SPaSM (Scalable Parallel Short-range Molecular-dynamics) du Laboratoire National de Los Alamos est une application massivement parallèle destinée à la simulation, l'analyse de données et la visualisation. Initialement monolithique, cette application a été modularisée puis mise sous le contrôle de python.

Le langage python a de nombreux avantages. Notamment, il permet un développement d'application rapide par sa simplicité et sa puissance. En effet, sa syntaxe peut être rapidement assimilée par des développeurs et son typage dynamique lui confère une grande efficacité. De plus, il est très extensible et permet d'utiliser du code en langages compatibles avec la pile d'appel du C. Il suffit pour cela de construire une librairie dynamique et de l'importer depuis l'interpréteur python comme n'importe quel module. Enfin, il est sous une licence compatible avec la GPL ce qui signifie que son code source est librement accessible et redistribuable. Ceci a l'avantage d'ouvrir des perspectives pour l'avenir de la plate-forme. On peut imaginer, par exemple, que l'application du projet CALVI soit complètement parallélisée afin de s'exécuter sur un *cluster*, et donc il y aurait peut-être intérêt à plonger au coeur du code de l'interpréteur python pour effectuer cette tâche.

Python offre donc une API en langage C (puisqu'il est implanté en C) qui permettrait d'atteindre l'un des objectifs : la modularité. A ce point, il nous faut encore choisir la méthode de construction des modules. On peut, par exemple, écrire leurs interfaces en utilisant les fonctions de cette API puis les compiler ensemble. Cependant, c'est une tâche assez fastidieuse et il est préférable d'utiliser des outils qui automatisent ce processus. Il en existe de nombreux et parmi ceux-ci nous pouvons tester les suivants :

Pyrex : programme permettant de mixer du code python et des types de données C pour compiler un module d'extension en C. Cela signifie que l'on peut écrire un module sous la forme d'un script python dans lequel on déclare des variables comme étant d'un type du C par `cdef type nom_variable` ; par exemple :

```
#
#  Calculate prime numbers
#

def primes(int kmax):
  cdef int n, k, i
  cdef int p[1000]
  result = []
  if kmax > 1000:
    kmax = 1000
  k = 0
  n = 2
  while k < kmax:
    i = 0
    while i < k and n % p[i] <> 0:
      i = i + 1
    if i == k:
      p[k] = n
      k = k + 1
      result.append(n)
    n = n + 1
  return result
```

(en ôtant les lignes **cdef**, on obtient le module python équivalent) ; alors pyrex traduit ce code en du code C et le compile en un module d'extension pour python ; ainsi, il n'est absolument pas nécessaire de connaître l'API Python/C pour développer un module d'extension. Cependant, il ne convient pas à nos attentes, car demanderait une réécriture du code existant.

Boost : librairie permettant d'écrire des modules d'extensions. A partir d'un code C++, on écrit un fichier contenant à la fois du code C++ et des directives propres à bjam, qui est un précompilateur fournit avec Boost ; par exemple pour la classe suivante :

```
struct World
{
   void set(std::string msg) { this->msg = msg; }
   std::string greet() { return msg; }
   std::string msg;
};
```

on écrit le fichier :

```
#include <boost/python.hpp>
using namespace boost::python;

BOOST_PYTHON_MODULE(hello)
{
   class_<World>("World")
      .def("greet", &World::greet)
      .def("set", &World::set)
   ;
}
```

bjam génère alors un code utilisant les fonctions de cette librairie et qui, compilé avec le code initial et lié avec Boost, donne le module ; L'inconvénient majeur est qu'ici le module résultant

doit être lié avec cette librairie ce qui impose qu'elle réside sur la machine exécutant le code. En outre, la syntaxe des directives de précompilation peut aussi représenter un obstacle.

Swig : programme générant le code d'interfaçage entre du code C/C++ et l'interpréteur python. Il scanne les fichiers d'en-tête spécifiés et crée le code C/C++ encapsulant celui existant. Il demande une intervention minimum du développeur et permet de créer rapidement un module à partir d'un code déjà présent ; des exemples seront donnés dans la partie suivante.

f2py : programme créant un module d'extension C à partir d'un code fortran. Il génère aussi une documentation du module et analyse le code fortran pour simplifier le code python appelant. Il peut prendre de manière optionnelle un fichier de signature déclarant la structure du future module python ou bien des directives inclues directement dans le code fortran ; par exemple le fichier source fortran suivant, contenant des commentaires (lignes débutant par cf2py) réservés à f2py, construit un module contenant la fonction **`mult`** réalisant le produit de deux matrices :

```
      SUBROUTINE MULT (A, m, n, B, o, p, C)
      INTEGER m, n, o, p, i, j, k
      INTEGER A(m,n), B(o,p), C(m,p)
cf2py intent(in) m, n, o, p, A, B
cf2py intent(out) C
cf2py depend(m) C
cf2py depend(p) C
      INTEGER t
      DO 320 i=1, m
         DO 310 j=1, p
            t = 0
            DO 300 k=1, n
               t = t + A(i, k) * B(k, j)
 300        CONTINUE
            C(i, j) = t
 310     CONTINUE
 320  CONTINUE
      END
```

Pyfort : similaire à f2py avec une documentation plus pauvre et une utilisation moins facile.

Ainsi, le meilleur choix concernant ces outils semble être swig pour les *solvers* écrits en C++ et f2py pour ceux en fortran. Il n'existe malheureusement pas un tel programme gérant à la fois ces deux langages.

3 Développement de la plate-forme

Cette section a pour but de poser les bases de la future plate-forme. Nous verrons donc en détail l'architecture logicielle sous jacente, l'API proposée ainsi que l'implantation du module de contrôle.

3.1 Architecture logicielle

Cette plate-forme se base sur l'interpréteur python qui se place donc au centre avec le module de contrôle. Autour gravite l'ensemble des autres modules constitués par :

– les paramètres physiques indépendants de la méthode numérique utilisée,
– les *solvers* vlasov,
– les *solvers* champs,
– la visualisation,
– l'export dans des fichiers de données.

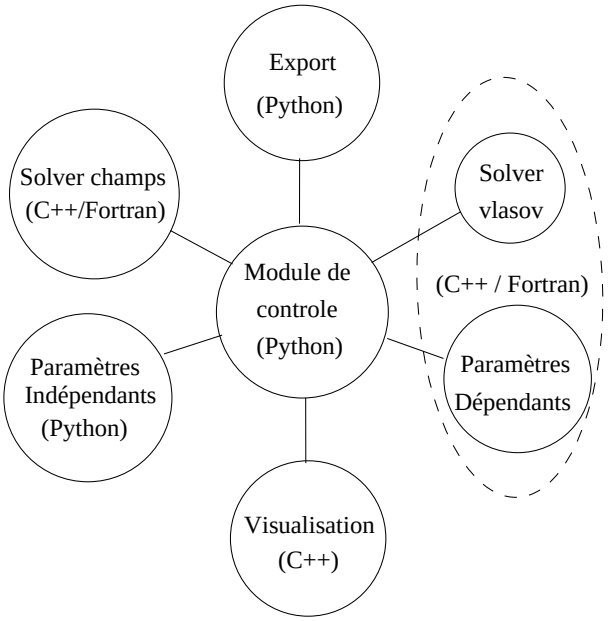


FIG. 2 – Architecture de la plate-forme

Le schéma précédent FIG. 2 représente cette architecture par un ensemble d'entités logiques reliées au module central. Dans la partie suivante, nous allons décrire l'API à laquelle nous avons abouti au cours de ce projet.

3.2 API

Cette API concerne un *solver* Vlasov 1D et un *solver* champ Poisson. Elle se base essentiellement sur le code `Obiwan`. Celui-ci intègre donc les deux parties de la résolution numérique. En outre, il permet l'enregistrement des résultats dans des fichiers binaires ou au format ASCII ainsi que la visualisation grâce à la librairie OpenGL. Ce programme étudie ainsi le problème physique à une dimension dans plusieurs cas de figure, déterminés par un ensemble de paramètres. Ceux-ci sont pris dans un fichier de configuration qui contient, en particulier, la nature de la distribution initiale, qu'on nommera "cas test" par la suite.

Dans un premier temps, il a fallu scinder en deux l'ensemble des paramètres du programme. Ils interviennent d'une part dans la méthode numérique utilisée, ce sont alors des paramètres physiques, d'autre part dans la visualisation et l'export. Nous avons donc établis la liste suivante :

- paramètres indépendants de la méthode utilisée

  - constantes physiques : π, ε_0, ...
  - pas de temps : dt
  - géométrie du domaine physique (pour le cas à une dimension) : x_{min}, x_{max}, v_{min}, v_{max}
  - export : ASCII ou binaire, nombre d'itérations, fréquence du diagnostic
- paramètres dépendants et spécifiques à `Obiwan`
  - maillage : j_x, j_v, $nblevel$
  - norme de seuillage et seuil
  - type d'ondelette utilisée (Haar, interpolette, ...)
  - cas test (cylindre, amortissement landau, faisceau gaussien, ...)

Ainsi, les paramètres indépendants donnent lieu à une classe à part entière gérée par le module de contrôle. Les autres font partie intégrante de la classe du *solver*.

La donnée la plus importante et nécéssitant d'être normalisée pour tous les *solvers* Vlasov est la fonction de distribution. Elle sera donc représentée par un tableau à deux dimensions, indicé en x pour les colonnes et en v pour les lignes, dont les cellules **`distrib[i][j]`** correspondent à la valeur de cette fonction pour une abscisse de $x_{\text{min}} + j \times dx$ et une vitesse de $v_{\text{min}} + i \times dv$, dx et dv étant respectivement la taille d'une maille en x et en v.

De même, les champs électriques (**`efield`**), de force (**`forcefield`**) et densité de particule (**`rho`**) seront représentés par des tableaux à une dimension, dans le cas considéré (Vlasov 1D).

Concernant l'API proprement dite, les modules *solver* Vlasov doivent fournir :

- un constructeur (ou une fonction d'initialisation) prenant en argument les paramètres indépendants ainsi qu'une chaine de caractère représentant le nom d'un fichier de configuration dans lequel seront stockés les paramètres spécifiques à ce *solver*.
- des accesseurs pour la distribution, de profil :

  ```
  getDistrib() : distrib
  setDistrib(d : distrib)
  ```

- et les méthodes suivantes, réalisant le calcul :

  ```
  computeRho(r : rho)
  advection(f : forcefield)
  ```
  ou dans le cas d'un splitting X/V :
  `advectionX(f : forcefield)` et **`advectionV(f : forcefield)`**

Les *solvers* champs doivent fournir :

- un constructeur (ou fonction d'initialisation)
- et les méthodes suivantes :

  ```
  computeEself(r : rho) : efield
  computeForce(Eself : efield, Eapplied : efield) : forcefield
  ```

  Enfin, la classe export, servant à écrire les résultats sur disque, propose les fonctions de diagnostic pour les deux types de *solver* :

  ```
  diagnosticVlasov(iter : integer, d : distrib)
  diagnosticField(iter : integer, r : rho, e : efield)
  ```

3.3 Implantation

Ayant travaillé uniquement sur « `Obiwan` », l'implantation actuelle ne concerne que du code C++. Ainsi, python a été utilisé pour implanter le module de contrôle et le module d'export, swig pour construire les autres modules.

3.3.1 Python

Le module de contrôle se compose de deux fonctions : la première (**`init`**) permet l'initialisation des différents paramètres et des *solvers*, la seconde (**`steps`**) constitue la boucle principale qui les appelle en leur transmettant les données sur un nombre déterminé d'itérations.

Le programme se découpe donc en trois phases :

– import des différents modules nécéssaires au déroulement du programme

```
import utils,param,vlasov,field,time,export
```

– initialisation ; la fonction **`init`** crée les objets **`solverVLASOV`** et **`solverFIELD`** à partir des modules correspondant, elle initialise les différents paramètres et les variables globales comme **`distrib`** pour stocker la distribution

```
def init(filename):
    global solverVLASOV, solverFIELD, diag, iter, nbiter
    global p, nblevel, dt, efield, rho, distrib [...]
    p = param.parameter(filename)
    solverVLASOV = vlasov.vlasovSolver()
    solverFIELD = field.force_field()
    diag = export.Export(p)
    [...]
```

– calcul ; la fonction **`steps`** boucle tant que le nombre d'itération total n'est pas atteint ; à chaque itération, elle appelle les méthodes d'advection du *solver* Vlasov ainsi que les différentes fonctions de calculs

```
def steps():
    global iter,nbiter,solverVLASOV,solverFIELD,efield,rho,dt,tn,p
    while iter < nbiter:
        start = time.clock()
        print "*************** Iteration",iter,"***************"
        solverVLASOV.splittingV(dt,efield)
        if (iter + 1) % p.get_ifreq() == 0:
            solverVLASOV.splittingX(0.5*dt, 1, efield)
            if p.do_diagnostic() != 0:
                solverVLASOV.getDistrib(distrib)
                diag.diagnosticVLASOV(iter, distrib)
            solverVLASOV.splittingX (0.5*dt, 0, efield)
        else:
            solverVLASOV.splittingX (dt, 0, efield)
        iter = iter + 1
        tn = tn + dt
        solverVLASOV.computeRho(rho)
        solverFIELD.computeField (tn, rho, efield)
```

```
19
20          finish = time.clock ()
21          durationTotal = finish - start
22          print "Temps total ",durationTotal
```

3.3.2 Swig

Pour ce qui est de la construction des modules, l'utilisation de swig est assez simple. Ce programme nécessite l'écriture de fichiers d'interface dans lesquels sont déclarés :

- le nom du module,
- les fichiers d'en-tête contenant la définition des classes et fonctions qui composeront le module,
- un certain nombre de directives facultatives permettant d'affiner la construction du module. Par exemple, **`%addmethods`** permet d'ajouter une méthode à une classe sans avoir à modifier le code C++ correspondant.

Voici, un tel fichier pour le module **`field`** :

```
1  %module field
2  %{
3  #include "field.h"
4  %}
5  %include field.h
```

En ce qui concerne la distribution, ce tableau a été modélisé par une liste de listes en python, ce qui correspond bien à la structure attendue. Cependant, cela demande une copie des valeurs à chaque diagnostic puisque la distribution réside à l'intérieur du *solver* vlasov. Cela n'a aucune répercution pour des domaines de taille réduite (les tests de performances le prouvent) mais en aura sans doute sur de grands domaines.

Le fichier d'interface swig pour le *solver* vlasov comporte donc la déclaration d'une nouvelle méthode (la fonction **`getDistrib()`** de l'API) qui permet la conversion du type de donnée utilisé en C++ à celui en python :

```
1  %module vlasov
2  %{
3  #include "vlasovsolver.h"
4  %}
5  %include vlasovsolver.h;
6  %typemap(python,in) PyObject *distrib {
7          $target = $source;
8  }
```

```
9  %addmethods vlasovSolver {
10    // adds methods getDistrib
11    PyObject *getDistrib(PyObject *distrib) {
12      int i,j;
13      PyObject * res;
14      PyObject * it;
15      int size1,size2;
16      const Array2d & fvalue = self->getFValue();
17      size1 = fvalue.getSize1();
18      size2 = fvalue.getSize2();
```

```
19      if (!PyList_Check(distrib)) {
20        distrib = PyList_New(size1);
21      }
22      if (PyList_Size(distrib) != size1) {
23        distrib = PyList_New(size1);
24      }
25
26      for (i=0; i < size1; i++) {
27        it = PyList_GetItem(distrib,i);
28        if (!PyList_Check(it)) {
29          it = PyList_New(size2);
30        } else if (PyList_Size(it) != size2) {
31          it = PyList_New(size2);
32        }
33        for (j=0; j < size2; j++) {
34          if (fvalue(i,j))
35          PyList_SetItem(it,j,PyFloat_FromDouble(fvalue(i,j)));
36          else {
37            if (PyFloat_AsDouble(PyList_GetItem(it,j)))
38            PyList_SetItem(it,j,PyFloat_FromDouble(0.0));
39          }
40        }
41      }
42      return res;
43    }
44  }
```

3.4 Perspectives

Le découpage d'`Obiwan` et son intégration à la plate-forme répondent bien à nos attentes. En d'autres termes, les résultats que l'on obtient lors d'une exécution sont strictement identiques à ceux du programme initial, la visualisation mise à part, celle-ci n'étant pas encore prise en charge. Ainsi, l'application obtenue est tout à fait fonctionnelle et conforme aux objectifs.

Cependant, n'oublions pas que l'objectif final à atteindre avec cette plate-forme est de rendre le plus transparent possible l'utilisation de plusieurs *solvers* simultanément. L'idée serait de pouvoir écrire un cas test physique en python, puis de le simuler en utilisant des *solvers* différents en terme de modélisation physique, de méthode numérique et d'implantation. Ceci permettra de confronter les résultats de manière plus direct, et donnera aux physiciens un outil de simulation plus souple qu'aujourd'hui. La plate-forme devra être améliorée en considérant les points suivants :

- L'API proposée ici devra être améliorée afin d'être la plus générale possible, c'est-à-dire totalement indépendante du type de *solver* utilisé. En effet, pour l'instant l'API répond correctement aux besoins du code Obiwan, mais des problèmes surviendront peut-être lors de l'intégration de nouveaux codes à la plate-fome.

- Tous les *solvers* ont été développés complètement indépendemment les uns des autres, et abordent différemment l'aspect physique de la simulation. Une uniformisation des unités (physiques) utili-

sées dans les *solvers* sera donc nécéssaire.

- Le module d'export devrait permettre d'envoyer les données à un programme d'analyse ou de visualisation externe, peut-être par l'intermédiaire de sockets, très bien gérées en python.

- Tout ce travail a été effectué dans le cadre du découpage d'un code séquentiel. Il faudra surement remettre en question certaines parties de ce travail lorsque des codes parallèles seront intégrés à la plate-fome. En particulier l'export actuel des résultats sur disque risquera de constituer un goulot d'étranglement.

- Il restera encore à trouver le bon compromis entre l'efficacité du code et la souplesse d'utilisation de la plate-forme.

4 Guide d'utilisation du langage Python

Ce guide de programmation en python n'a pas pour vocation de vous montrer l'étendue des possibilités qu'offre ce langage, mais de vous en donner les bases. Ainsi, vous pourrez très rapidement commencer à développer vos propres programmes. Les domaines d'application sont nombreux, de même que les librairies fournies avec le système ou téléchargeables sur internet. Il est vivement conseillé de consulter les adresses de sites *Web* données en annexes (cf. Annexe E. Ressources p. 50) si vous souhaitez approfondir vos connaissances en Python.

Python a été créé en 1990 par Guido VON ROSSUM. Son nom provient de la série anglaise *Monthy Python's Flying Circus*, de laquelle l'auteur est un fan. En 1991, Il livre les sources de ce langage au domaine public. En 1994, le *newsgroup* `comp.lang.python` est ouvert. Il permet aux développeurs en nombre croissant de s'entraider. Un an plus tard, le site officiel de Python est créé et des associations naissent autour de ce langage : la PSA (Python Software Activity) aide à la promotion de Python par le développement de sites et l'organisation de conférences. Le « Python Consortium » lui apporte une aide financière, il est composé d'entreprises privées désirant investir dans son développement. La version actuellement disponible en mai 2004 est la version 2.3.

Simplicité, portabilité et extensibilité font parties des qualités de python. Voyons ces caractéristiques un peu plus en détail :

simplicité – Sa syntaxe est très facile à apprendre et, combinée à des types de données évolués, permet de rapidement mettre en oeuvre ses atouts dans des scripts compacts mais lisibles.

étendue – Avec sa librairie très fournie, Python a accès à une grande variété de services tels que les appels systèmes UNIX, les protocoles internet, les bases de données, les interfaces graphiques ...

extensibilité – Il a la capacité de s'interfacer avec des librairies écrites dans différents langages tels C, C++ ou fortran, ce qui lui permet d'être au coeur de grands projets scientifiques.

portabilité – Il peut s'exécuter sur les différentes implémentations d'UNIX, de même que sur MacOS, Windows et autres systèmes d'exploitation.

orienté objet – Il possède tous (ou presque[1]) les outils modernes de la programmation orientée objet (POO) tels que l'héritage multiple, le polymorphisme, la surchage d'opérateur, la gestion des *exceptions*, etc.

haut niveau – Il possède un typage dynamique, déterminé à l'exécution par son puissant interpréteur de commande, de même que des outils comme un *garbage collector* gérant de manière automatique la mémoire allouée. Il est, de plus, réflectif, c'est-à-dire qu'il permet à des objets d'être modifiés en cours d'exécution (p.ex. ajout d'attributs et méthodes, ou encore changement de classe).

open source – Il est en libre téléchargement, sur de nombreux sites internet, en particulier le site officiel (http ://www.python.org). Il est distribuable gratuitement et peut être intégré dans n'importe quel logiciel selon les termes de la GPL[2].

Ses nombreuses qualités, permettent à Python d'avoir de nombreuses applications, comme les scripts systèmes, les scripts *cgi* pour internet, les logiciels scientifiques, les interfaces graphiques, etc.

4.1 Première approche

Dans cette section, nous verrons comment exécuter un programme python puis un premier exemple simple.

4.1.1 Interpréteur python

Comme pour de nombreux langages, les programmes python se présentent sous la forme de fichiers texte (au format ASCII), généralement avec l'extension *.py*. Ils sont compilés en *bytecode* puis exécutés par l'interpréteur du même nom : `python`.

```
$ python helloworld.py
Hello World!
```

Celui-ci peut aussi être invoqué de manière interactive (option `-i`) de sorte que l'on est en présence d'un *prompt* (par défaut `>>>`) qui accepte en ligne de commande des instructions python.

```
$ python -i
>>> print 'Hello World!'
Hello World!
>>> Ctrl-D[3]
```

Ces deux modes peuvent être combinés afin qu'après l'exécution d'un script on ait encore accès aux variables, fonctions, classes et objets qui y ont été créés.

```
$ python -i helloworld.py
Hello World!
>>> print str
Hello World!
>>> Ctrl-D
```

[1] Le seule manque à ce niveau concerne les permissions d'accessibilité des méthodes et attributs qui sont toujours publiques.
[2] General Public License (cf. le site de la Free Software Fundation : http ://www.fsf.org)
[3] Ctrl-D permet de quitter l'invite de commande python.

4.1.2 Structure d'un Programme

Un programme python se compose de plusieurs parties qui peuvent être classées en trois catégories :

commentaires : commencent par le caractère # et se terminent à la fin de la ligne. À noter que dans un environnement UNIX, la première ligne de commentaire permet de préciser l'interpréteur à appeler lorsque le script est exécuté directement.

déclarations : elles permettent de définir des fonctions et des classes à l'aide du mot clé `def`. Les variables ne possèdent pas de type propre puisqu'il est évalué dynamiquement. Elles sont donc déclarées uniquement par une affectation.

instructions : elles sont la partie exécutable d'un script python. Elles sont composées d'appels de fonctions, de structures conditionnelles telles que les tests `if`, les boucles `for`, `while`, ... Lorsqu'elles sont dans la définition d'une fonction ou d'une classe elle ne sont exécutées respectivement qu'à l'appel de la fonction ou à la création d'un objet. Sinon, elles sont exécutées une seule fois lors de l'interprétation du script.

Un programme python très simple ressemble donc à celui-ci :

```
1 #!/usr/bin/python
2 # programme helloworld
3
4 str = 'Hello World!'
5 print str
```

A l'exécution, l'interpréteur crée un objet de type chaîne contenant la valeur `'Hello World!'`, ainsi qu'une référence (non typée) portant le nom `str` et pointant sur cet objet. L'instruction `print` imprime à l'écran une ou plusieurs chaînes de caractères séparées par des virgules et placées à sa suite sur la ligne.

⚠ Les variables sont donc des références qui peuvent à tout instant, lors de l'exécution d'un programme python, pointer sur des objets de types différents. Ainsi, il est tout à fait correct, en python, d'écrire dans un script :

```
str = 'Hello World!'  str = 3.0
```

Nous verrons dans les sections suivantes les types de données et la syntaxe de python plus en détail.

4.2 Types et Structures de données

Comme vu précédemment, python a un typage dynamique mais cela ne signifie pas qu'il n'y a pas de types en python, cela implique seulement qu'une variable est en réalité une référence ne possèdant pas de type mais l'objet qu'elle pointe en revanche en a un.

Il y a plusieurs types de données simples tels les nombres et les chaînes, et d'autres extrêmement évolués permettant de satisfaire la majeure partie des besoins en programmation tels les listes, les tuples et les dictionnaires. De plus, ceux-ci peuvent aussi être séparés en deux catégories : modifiables ou non.

En effet, certaines structures de données ne peuvent pas être modifiées aprés leur création, cela permet d'assurer leur intégrité.

Enfin, nous ne parlerons pas encore de la possibilité de créer de nouveaux types par la définition de classes qui sera l'objet de la section 4.5 page 27.

4.2.1 Types simples

nombres

Python offre quatre types de données numériques :

entier pour les nombres entiers compris entre -2147483648 et 2147483648 ;

entier long pour les nombres entiers avec une précision illimitée[4] ; les entiers longs sont représentés par une suite de chiffres se terminant par un `L` (p.ex. `n = 2200000000L`) ;

réel pour les réels en double précision à virgule flottante ; les déclarations suivantes sont des réels corrects en python : `r1 = 2.5, r2 = 4.1e6, r3 = 4.1E6` ; la lettre e est ici pour préciser l'exposant d'un facteur 10 ;

complexe pour les nombres complexes avec `j` comme nombre imaginaire (p.ex. `c = 5 + 3j` ou encore `c = complex(5,3)`) ; ils sont représentés par un couple de réels ; ces objets ont les attributs `real` et `imag` qui permettent d'extraire respectivement leur partie réelle ou imaginaire (p.ex. `c.imag`) ;

Les opérateurs autorisés dans une expression avec des nombres sont `+`, `-`, `*` et `/` qui ont la signification habituelle. Il y a de plus l'opérateur d'exponentiation `**` pour la mise en exposant.

Enfin, il est possible d'utiliser l'interpréteur python comme un simple calculateur. En effet, lorsqu'une expression n'est pas affectée à une variable, le résultat est imprimé à l'écran, par exemple :

```
>>> 256.34 * 89 + 10.7
22824.959999999999
>>> 256.34 * 89 + 10.7e45
1.07e+46
>>> complex(5,3) * 5.2 + 54j
(26+69.599999999999994j)
>>> (5 + 3j) * 5.2 + 54j
(26+69.599999999999994j)
>>> 58346987 * 895462133
52247517433143271L
>>>
```

Remarquez le dernier résultat : il dépasse les limites des entiers et est donc converti en entier long.

chaînes

[4] la précision est en fait limitée uniquement par la quantité de mémoire virtuelle accessible sur la machine.

Le type chaîne permet de manipuler des chaînes de caratères. Les simples (') et doubles (") *quotes* sont utilisées pour délimiter les chaînes. On peut de plus avoir recours aux triples (""") *quotes* qui autorise la saisie d'une chaîne sur plusieurs lignes, p.ex.

```
str = 'Une chaîne simple'
str = "Une autre chaîne simple"
str = """Une chaîne
s'etalant sur plusieurs
lignes. """
```

Elles peuvent être formatées à l'aide de l'opérateur **%** : à l'intérieur de la chaîne, il précise le format (p.ex. **%d** et **%f** pour respectivemet les entiers et les réels) ; à la suite de la chaîne, il précise les sources des substitions (des constantes ou des variables) qui doivent être du même nombre et type que les formats et entourées de parenthèses s'il y en a plus d'une.

```
>>> str = "Hello %s" % "World!"
>>> print str
Hello World!
>>> print 'un caractère : %c' % 'A'
un caractère : A
>>> print 'un entier : %d' % 1
un entier : 1
>>> print 'un réel : %f' % 1
un réel : 1.000000
>>> print 'des réels : %e %E' % (1.59e12, 1.59e12)
des réels : 1.590000e+12 1.590000E+12
>>> var = 1.59e12
>>> print 'des réels : %g %G' % (1.59e12, var)
des réels : 1.59e+12 1.59E+12
>>>
```

D'autres opérateurs utiles existent pour les chaînes, il s'agit de ***** qui permet de répéter une chaîne autant de fois que l'opérande droit et **+** pour la concatenation. La fonction **len** renvoie la longueur d'une chaîne.

```
>>> str = 'Hello'*5
>>> print str
HelloHelloHelloHelloHello
>>> str = 'Hello'
>>> str + 'World!'
'HelloWorld!'
>>> len(str)
5
```

Les chaînes sont des objets non-modifiables, les opérations **%**, **+** et ***** créent donc un nouvel objet lorsqu'elles sont évaluées.

Enfin, elles fonctionnent comme des listes et possède donc les mêmes attributs et opérations. Il est possible par exemple d'extraire une sous-chaîne ou un caractère en donnant l'indice ou les bornes entre crochets : **`str[2], str[3:6]`** .

4.2.2 Listes

Les listes sont des données composées qui permettent de regrouper d'autres données de tout type. Elles sont déclarées entre crochets et leurs éléments sont séparés par des virgules. On accède à chacun d'entre eux par son indice en commençant par 0. On peut aussi récupérer une sous-liste à l'aide d'un *slice*, c'est à dire une tranche de la liste spécifiée par ses bornes inférieures et supérieures séparées par deux points. Lorsqu'une des deux bornes manque, elle est mise au maximum. Enfin, ces bornes peuvent être négatives, auquel cas elles indicent à partir du dernier élément. Ainsi, **`l[2:5]`** est la liste l restreinte aux éléments $2, 3, 4 et 5$; **`l[:-2]`** est la liste l sans ces deux derniers éléments.

Les opérateurs, vus précédemment pour les chaînes, de concaténation (+) et répétition (*) sont aussi valables, de même que la fonction **`len`**. Les listes ont aussi une méthode **`append`** qui permet d'ajouter un élément en fin de liste.

```
>>> a = ['zero', 'un', 100, 1234] # on déclare a
>>> a[2] = a[2] + 23 # on remplace a[2] par sa valeur + 23
>>> a
['zero', 'un', 123, 1234]
>>> a[0:2] = [1, 12] # on remplace les 2 premiers éléments
>>> a
[1, 12, 123, 1234]
>>> a[0:2] = [] # on enlève les deux premiers éléments
>>> a
[123, 1234]
>>> a[1:1] = ['test', 'xyzzy'] # on insère à la position 1
>>> a
[123, 'test', 'xyzzy', 1234]
>>> len(a) # longueur de a
4
>>> a * 2
[123, 'test', 'xyzzy', 1234, 123, 'test', 'xyzzy', 1234]
>>> a.append(3.5)
>>> a
[123, 'test', 'xyzzy', 1234, 3.50000]
```


Si vous déclarez une liste **`l`** et faites **`lbis = l`** vous aurez alors deux références à la même liste ce qui signifie que si vous modifiez **`l`**, **`lbis`** le sera aussi. Si vous désirez obtenir l'*état* de **`l`** à cet instant, il faut forcer une copie de cette liste en faisant : **`lbis = l[:]`**.


Lorsque vous déclarez que l'élément n d'une liste l est l'objet o, vous ne stockez en fait qu'une référence à cet objet dans la liste et tout modification de celui-ci sera visible sur **`l[n]`**.

4.2.3 Tuples

Les tuples sont des listes non modifiables. Ils peuvent, comme les listes, contenir tout types d'objets python et ont accès aux mêmes opérateurs qu'elles. Ils sont représentés entre parenthèses et non entre crochets.

```
>>> a = ('test', 3, 5) # déclaration
>>> a
('test', 3, 5)
>>> len(a) # longueur
3
>>> a[0] # élément
'test'
>>> a[1] = 9 # une exception est levée
Traceback (most recent call last):
   File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

4.2.4 Dictionnaires

Les dictionnaires correspondent à des tableaux de hachage indexés par tout type d'objet non modifiable (nombres et chaînes essentiellement). Ils sont déclarés à l'aide d'accolades, chaque élément est composé d'une clé et d'une valeur séparées par deux points. Par Exemple dans l'expression suivante, `d = { 'chaîne': 56, 2.0 : 89 }` , on a `d['chaîne']` qui vaut 56 et `d[2.0]` qui vaut 89.

La fonction **len** renvoie le nombre d'éléments du dictionnaire. **keys** est une méthode de ces objets et renvoie la liste des clés et **has_key** teste l'existence d'une clé.

```
>>> d = {'chaîne': 56, 2.0: 89} # déclaration
>>> d['autre'] = 4127 # on ajoute un élément
>>> d
'autre': 4127, 'chaîne': 56, 2.0: 89
>>> d['chaîne'] # élément indicé par 'chaîne'
56
>>> d.keys() # liste des clés
['autre', 'chaîne', 2.0]
>>> d.has_key('autre') # autre est une clé donc vrai (=1)
1
```

4.3 Syntaxe de python

Cette section expose les éléments de syntaxe[5] les plus utiles en python.

[5] La grammaire complète du langage python est disponible à l'adresse http ://docs.python.org/ref/ref.html.

4.3.1 Remarques générales

La première remarque à noter est que le langage python possède une syntaxe qui impose la clarté dans le code d'un script. En effet, comme nous allons le voir, l'indentation des instructions est la base de sa structure de bloc et le retour chariot est souvent utilisé comme séparateur.

Les instructions exécutables de python sont simples ou composées. Les premières résident sur une seule ligne et se terminent par un retour chariot ('\n' en langage C). Il est possible de placer plusieurs instructions simples sur une ligne en les séparant par un point-virgule mais cela n'est pas recommandé pour la clarté du programme. Les secondes structurent le script en blocs d'instructions. Elles se composent d'un *en-tête* se terminant par deux points et d'une suite d'instructions indentées. En effet, au contraire d'un langage comme le C, il n'y a pas de balises telles que les accolades pour marquer les blocs. Ils sont délimités par leur indentation par rapport au bloc englobant.

 Les instructions de plus haut niveau dans cette hiérarchie de blocs sont celles à la *racine* du script. Elles ne doivent donc pas du tout être indentées sinon python provoque une erreur à l'exécution.

4.3.2 Instructions simples

affectation

L'affectation est la seule manière de déclarer une variable. Le signe **=** est utilisé ici. Il est possible d'opérer une déclaration multiple en séparant chaque élément par une virgule de même que chaque valeur. Cela correspond en fait à une identification entre deux tuples.

```
>>> x = 1 # affectation et déclaration de x
>>> x
1
>>> x, y, s = 1, 2, 'chaîne' # affectation multiple
>>> t = (1, 2, 'chaîne') # un tuple
>>> x, y, s = t # autre manière
```

print

C'est une fonction qui permet d'afficher sur la sortie standard des chaînes ou tout objet pouvant être converti en chaîne. Elle admet un nombre variable d'arguments séparés par des virgules (les virgules sont alors changés en espace et le tout est concaténé en une seule chaîne) . Les chaînes peuvent contenir des caractères d'échappement tels que **\n** ou encore **\t**.

```
>>> str = 'chaîne'
>>> print 2.0, str, 5 # print avec types mélangés
2.0000 chaîne 5
>>> print 'un = %d' % 1, 'et deux = 2' # utilisation d'un format
                                       # pour la première chaîne
un = 1 et deux = 2
```

import

Cette fonction permet d'importer un module python et donc d'avoir accès aux variables, fonctions et classes qui y sont déclarés. Leur nom est alors préfixé du nom du module suivit d'un point (p.ex. **`sys.ps1`** référence la variable **`ps1`** du module **`sys`**). Elle a la forme `import module1,module2,...`

Une autre forme d'import permet de rapatrier les noms dans l'espace de nom global (ang. global namespace) de sorte qu'il ne soit plus nécéssaire de les préfixer, il s'agit de `from module import *` pour tous les noms définis dans **`module`**, ou `from module import nom1, nom2, ...` pour seulement certains.

Plus de détails sur les espaces de noms sont donnés dans la section 4.7 page 34.


La dernière forme d'import présentée est à utiliser avec précaution. En effet, s'il existe dans l'espace de nom global les mêmes noms que dans le module, ils sont tout simplement écrasés et donc plus accessibles.

break et continue

Ces instructions n'ont pas de paramètres. Elle fonctionnent comme en langage C, et sont donc utilisées dans des boucles conditionnelles pour effectuer un branchement.

del

Cet instruction permet de détruire une ou plusieurs références (c'est à dire des variables). L'objet référencé décrémente alors son compteur de référence et s'il vaut 0, le *garbage collector* libère la mémoire utilisée. p.ex. `del x, y` .

Elle peut aussi être utilisée pour effacer un ou plusieurs éléments d'une liste ou d'un dictionnaire. p.ex. `del l[:2], d['chaîne']` .

return

C'est l'instruction qui permet aux fonctions et méthodes de renvoyer une valeur. Elle prend un nombre variable de paramètres. S'il y en a plus d'un, l'objet retourné est un tuple et peut être affecté à une ou plusieurs variables comme vu dans la section *affectation*. p.ex. `return a,b,c` .

global

Elle est utilisée dans un module, une fonction ou une classe pour préciser que les noms de variables en paramètres appartiennent à l'espace de nom englobant (cf. Section 4.7). p.ex. `global var1,var2`.

pass

Cette instruction ne fait rien mais est utile dans des cas où une instruction est attendue mais qu'on ne souhaite rien faire. p.ex. dans un bloc `except` pour la gestion des exceptions.

4.3.3 Structures de controle


Pour ces structures de code, il faut toujours faire attention à ce que l'indentation soit correct. Un conseil : utiliser un éditeur qui reconnait le langage pyhon et gère tout seul la bonne indentation (Emacs).

if

Le test conditionnel `if` a la structure suivante :

```
if condition1:
    instruction1
    instruction2
    ...
elif condition2:
    instruction1
    instruction2
    ...
else:
    instruction1
    instruction2
    ...
```

où les clauses `elif` et `else` sont optionnelles.

Les conditions sont des expressions booléennes pouvant être des comparaisons (`> >= < <= == !=`) et utilisant les opérateurs `or`, `and` et `not`. Une liste vide, une chaîne vide, 0 ou l'objet `None`[6] représentent la valeur *faux*, tout autre objet est considéré comme *vrai*.

while

Une boucle `while` se présente comme suit :

```
while condition:
    instruction1
    instruction2
    ...
else:
    instruction1
    instruction2
    ...
```

[6] Cet objet a une signifi cation particulière, en plus d'être le booléen *faux*, il est renvoyé par toute fonction qui ne renvoie rien ! Il posséde à peu près la même sémantique que le `NULL` en langage C.

La clause `else` est optionnelle. Si elle est présente, elle est toujours exécutée, à partir du moment où la condition est fausse.

for

Une boucle `for` est de la forme :

```
for var in liste:
    instruction1
    instruction2
    ...
else:
    instruction1
    instruction2
    ...
```

où liste peut être n'importe quelle liste d'objets même hétérogène. Python fournit les fonctions génératrices **`range([start,] stop[, step])`** et **`xrange([start,] stop[, step])`** [7] qui renvoient une liste d'entiers. Elles prennent un, deux ou trois paramètres entiers : la borne initiale, la borne finale et le pas ; le premier et le dernier étant optionnels (par défaut, start vaut 0 et step 1).

try

Cette structure permet d'attraper une exception lorsqu'elle est émise dans un bloc de code. Il existe deux formes pour cette instruction :

```
try:
    instruction1
    instruction2
    ...
except exception1:
    instruction1
    instruction2
    ...
else:
    instruction1
    instruction2
    ...
```

où il faut au minimum un bloc `except` et où le bloc `else` est optionnel. Ce dernier, s'il est présent, est exécuté si aucune exception n'a été levée.

[7] La seule différence entre ces deux fonctions est la construction de la liste. Avec range, elle est construite une fois pour toute au début de la liste, alors qu'avec xrange, elle est dynamique et l'élément courant est évalué à chaque entrée dans la boucle. C'est utile pour des listes très grandes.

Si une exception survient dans le bloc `try`, une clause `except` correspondante est recherchée. Si elle est trouvée, le code suivant est exécuté, sinon, l'exception est transmise au bloc englobant jusqu'à ce qu'elle soit captée par un autre bloc `try` ou bien par l'interpréteur python.

```
try:
    instruction1
    instruction2
    ...
finally:
    instruction1
    instruction2
    ...
```

Dans ce cas de figure, le bloc `finally` est toujours exécuté, quoiqu'il arrive. Si une exception est levée dans le bloc `try`, elle est retransmise au bloc englobant après l'exécution du code suivant `finally`.

4.4 Fonctions

Les fonctions, en python, font partie de la catégorie des objets *appelables* (ang. *callable*). Ce sont donc des objets à part entière et on peut créer des références pointant sur eux. *appelable* signifie qu'elles peuvent admettre des paramètres et exécutent du code.

4.4.1 Déclaration

On crée une fonction de la manière suivante :

```
def nom ( params ):
    " description "
    instruction1
    instruction2
    ...
```

Le mot clé `def` a la même sémantique que le signe `=`. Il sert à déclarer une référence (ici `nom`) sur un objet de type fonction constitué par le corps de cette fonction. Le corps est le bloc qui commence à la ligne suivante et doit être indenté. De manière optionnelle, il peut commencer par une chaîne constituant une documentation de cette fonction. Elle est accessible directement depuis un programme python (ou depuis l'interpréteur) par l'attribut `__doc__` de cet objet. Par exemple, pour la fonction `abs` de la librairie standard, on obtient la documantation suivante :

```
>>> print abs.__doc__
abs(number) -> number

Return the absolute value of the argument.
```

Les fonctions possèdent un espace de nom local. Cela implique que toute affectation crée une variable dans cet espace et non dans l'espace de nom global (à moins d'avoir été préalablement indiqué à l'aide de l'instruction `global`). Cependant, pour une variable référencée dans la fonction, python consulte la table des symboles locale, puis globale, puis celle des noms prédéfinis (ang. *built-in names*). Pour plus d'informations, consultez la section 4.7 page 34.

Une fonction ainsi définie est exécutée à chaque fois qu'elle est appelée, ce qui se fait de manière simple : `nom(params)` . Il est possible d'affecter à une variable un objet fonction et de s'en servir par la suite comme si elle avait été déclarée avec ce nom. On peut alors exploiter cette propriété dans des constructions comme les suivantes :

```
# affectation d'un nouveau nom et appel
autre_nom = nom
autre_nom(params)

# utilisation dans une boucle for
# func1, func2, func3 etant deja definies
for f in (func1, func2, func3):
    f()
```

Une fonction peut ou non renvoyer une valeur à l'aide de l'instruction `return` vue dans la section précédente.

4.4.2 Paramètres

La liste des paramètres d'une fonction est simplement une liste de noms séparés par des virgules qui seront utilisés dans le corps. Par exemple :

```
def quotient(a,b): return a/b
```

Ces paramètres sont passés *par valeur* à la fonction, mais cette valeur est toujours une référence d'objet et non la valeur de l'objet en question.

Il est possible de fournir une valeur par défaut à certains arguments, ainsi lors d'un appel à la fonction, on peut omettre de les citer tous. La construction est la suivante :

```
def nom ( param1, param2=valeur1, param3=valeur2):
   ...
```

La fonction précédente accepte donc au moins un paramètre (le premier) et au plus trois.

Ceci est à différencier avec la possibilité d'appeler une fonction avec des *mots clés* en argument. En effet python permet de passer ses paramètres à une fonction dans un ordre arbitraire à la condition de citer le nom de l'argument. Ainsi, la fonction précédente pourrait être appelée des manières suivantes :

```
nom(1.0)
nom(param2 = 'chaîne', param1 = 1.0)
nom(1.0, param3 = 2)
nom(1.0, 'chaîne', 2)
```

Ensuite, une fonction peut admettre un nombre variable de paramètres. Pour cela, il faut que le dernier paramètre dans sa déclaration soit précédé du symbole `*`. Auquel cas, ce paramètre recevra un tuple contenant la liste des arguments supplémentaires reçus. Par exemple, la fonction `fprintf` du langage C pourrait être déclarée et appelée ainsi :

```
def fprintf(file, format, *args):
    file.write(format % args)

fprintf(fichier, 'Renseignements :\n')
fprintf(fichier, 'Nom : %s Age : %d\n', 'Durand', 26)
```

4.4.3 Fonctions anonymes

Enfin, avec python on peut définir des petites fonctions anonymes à l'aide du mot clé `lambda`. Elle sont restreintes à une seule expression et peuvent être utilisées partout où un objet fonction est attendu. p.ex. `lambda a, b: a+b` est une fonction retournant la somme de ses deux paramètres.

4.5 Programmation orientée objet

La programmation orientée objet (POO) est un aspect important du langage python. Il permet d'organiser les données, réutiliser du code existant et exploiter toutes les propriétés de ce type de programmation.

4.5.1 Classes

La classe est la base de la POO. Elle a pour but d'offrir un nouveau type de données possédant des attributs et des méthodes. Il y a deux manière de définir une classe en fonction de sa situation dans la *hiérarchie de classe*.

définition
Voici une définition de classe simple :

```
class MaClasse:
    " description "
    instruction1
    instruction2
    ...
```

Comme d'autres objets, elle peut avoir une documentation sous forme de chaîne de caractères placée sur la première ligne suivant l'en-tête. Les instructions qui la composent (et qui sont, je le rappelle, toujours indentées) peuvent être toute instruction déjà vue. En particulier, une affectation de variable ou une définition de fonction lui associe des attributs et méthodes. Une classe possède son propre espace de nom. Ainsi variables et fonctions qui y sont crées lui appartiennent.

héritage

Lorsqu'une classe hérite d'une ou plusieurs autres, elle doit être définie de la manière suivante :

```
class ClasseFille (Mere1, Mere2, ...):
    instruction1
    instruction2
    ...
```

où **Mere1** et **Mere2** sont les classes parentes de celle-ci. Lorsqu'elle hérite, une classe récupère attributs et méthodes de ses *super classes*.

 La convention adoptée par python pour la recherche d'une référence dans la classe est de, d'abord, regarder dans son corps. Si elle n'est pas trouvée, alors python regarde dans la liste des *classes mères* en commençant par la plus à gauche dans sa déclaration.

4.5.2 Objets

De même qu'une fonction, une classe est un objet appelable. Son rôle, lorsqu'elle est appelée, est de créer une instance de la classe.

Une instance se déclare en appelant le nom de la classe suivit d'une paire de parenthèses avec ou non des paramètres en fonction du constructeur :

```
monobj1 = Classe1()
monobj2 = Classe2(1.0, 'chaîne')
```

Un objet du type défini par la classe est alors créé ainsi qu'une référence pointant sur celui-ci.

Lorsqu'une classe est appelée, la méthode **__init__**, si elle existe, est appelée. C'est le constructeur et il permet de procéder à l'initialisation de variables par exemple.

On accède aux attributs et méthodes d'un objet en les préfixant du nom de cet objet suivi d'un point :

```
monobj.x
monobj.f()
```

Les méthodes déclarées dans la classe doivent avoir au moins un paramètre. C'est une référence à l'objet appelant, premier argument de la fonction généralement appelé **self**. Il correspond au **this** du C++.

4.5.3 Exemple

Voyons cela sur un exemple :

```
1  # on import arctangente et racine du module math
2  from math import sqrt,atan
3
4  # définition d'une classe
5  class Point:
6      "modélisation d'un point du plan"
7
8      # le constructeur prend deux arguments supplémentaires
9      def __init__(self, x, y):
10         self.abscisse = x
11         self.ordonnee = y
12
13     # une méthode
14     def cart2pol(self):
15         r = sqrt(abscisse ** 2 + ordonnee ** 2)
16         theta = atan(ordonnee / abscisse)
17         return r, theta
```

```
18 # définition d'une autre classe qui en hérite
19 class Rectangle(Point):
20     """modélisation d'un rectangle donné par:
21          la position de son coin supérieur gauche
22          sa longueur et sa largeur"""
23
24     def __init__(self, x, y, L, l):
25         self.abscisse = x
26         self.ordonnee = y
27         self.longueur = L
28         self.largeur = l
29
30     def surface(self):
31         return longueur * largeur
32
33 # utilisation
34 p = Point(1,2)
35 print "coord. cart. : p = (%f,%f)" % (p.abscisse, p.ordonnee)
36
37 rayon, angle = p.cart2pol()
38 print "coord. pol. : p = (%f,%f)" % (rayon,angle)
39
40 r = Rectangle(2,2,5,6)
41 print "surface de r : %f" % r.surface()
```

On remarque que dans le constructeur de la classe **`Point`**, lignes 10 et 11, nous avons défini deux variables (**`abscisse`** et **`ordonnee`**) dans l'objet **`self`**. Ces variables deviennent donc des attributs de cette classe. À la ligne 15, la méthode **`cart2pol`** utilise ces attributs, elle n'a pas besoin de les préfixer de **`self`** puisqu'ils sont dans l'espace de nom associé à la classe. L'appel de méthode d'un objet se fait aussi sans préciser **`self`** qui est implicite. En fait **`r.surface()`** est équivalent à **`Rectangle.surface(r)`**, c'est à dire en passant directement par l'objet `class`.

4.6 Modules

Un *module* est un objet python contenant du code (définitions, déclarations, instructions) et permettant sa réutilisation. C'est généralement un *script* mais comme nous le verrons, il est possible de définir des modules sous forme de code exécutable comme des bibliothèques dynamiques écrites en langage C par exemple.

4.6.1 Définition

La manière la plus simple et la plus courante de définir un module est de créer un fichier texte : un script. Celui-ci *doit* impérativement avoir l'extension .py afin qu'il soit reconnu par l'interpréteur (cf. Utilisation). Il peut contenir toutes les instructions python vues précédemment. Notamment, un module contient généralement des définitions de fonctions et de classes. Lorsqu'il est chargé pour la première fois, python le compile en *bytecode* et le sauvegarde sous cette forme dans un fichier du même nom portant l'extension .pyc, de sorte que lors d'une utilisation ultérieure, le chargement soit plus rapide. Ce fichier reste indépendant de l'architecture et n'est utilisé que si sa date de création est plus récente que le script lui-même[8].

Lorsqu'un module contient des instructions exécutables, celles-ci sont évaluées lors de la première importation, mais si vous réimportez ce module dans une même session, elles ne le sont pas. Comme tout objet python, un module peut posséder une description stockée dans une chaîne de caractères, accessible par la variable **`nom_module.__doc__`**. Elle doit précéder toute autre instruction. Nous définissons donc, à titre d'exemple, le module suivant :

```
1  # fichier textproc.py
2
3  """Le module textproc fournit les fonctions listing
4  et grep pour le traitement de fichiers texte."""
5
6  # fonction prenant en argument une chaîne, nom de fichier
7  def listing(filename):
8      """fonction listing: affiche le contenu d'un fichier texte
9      en numérotant les lignes."""
10
11     textfile = file(filename, mode='r')
12     lineno = 1
13     for line in textfile.xreadlines():
14         print '%d:' % lineno, line[:-1]
15         lineno = lineno + 1
16     textfile.close()
```

[8] Si vous modifiez votre module, il est inutile d'effacer ce fichier, car il sera automatiquement recompilé.

```
17 # fonction prenant en argument un nom de fichier
18 # et une chaîne à rechercher
19 def grep(filename,pattern):
20     """fonction grep: recherche un motif dans un fichier
21     et affiche la ligne correspondante s'il est trouvé"""
22
23     textfile = file(filename, mode='r')
24     lineno = 1
25     found = None
26     for line in textfile.xreadlines():
27         if line.find(pattern) != -1:
28             print '%d:' % lineno, line[:-1]
29             found = 1
30         lineno = lineno + 1
31     if not found:
32         print pattern, "not found in file", filename
33     textfile.close()
```

L'interpréteur python a aussi la capacité de charger dynamiquement des fichiers objets [9], compilés à partir de langages comme C, C++ ou encore fortran. Ils doivent utiliser l'API python mais leur construction peut être simplifiée grâce à l'utilisation d'outils comme *swig* ou encore *f2py*. Plus d'informations à ce sujet sont données dans l'annexe D page 41.

4.6.2 Utilisation

Le nom d'un module est le nom du fichier qui le défini et auquel on a retiré l'extension. Un module se charge en mémoire à l'aide des instructions `import nom_module` et `from nom_module import` (cf. section 4.3 page 20). L'interpréteur recherche alors un fichier **`nom_module.py`** ou **`nom_module.so`** dans le répertoire courant. S'il ne le trouve pas, il recherche dans une liste de répertoires spécifiée par la variable d'environnement **`$PYTHONPATH`**[10].

Lorsqu'un module est importé, on peut alors accéder aux objets qui y sont définis en les préfixant du nom de celui-ci. Dans l'exemple qui suit, nous allons utiliser le module définit à la section précédente :

```
>>> import textproc
>>> textproc.listing('tmp.txt')[11]
1: I am not a citizen, national or resident of, and am not
2: under the control of, the government of: Cuba, Iran, Sudan,
3: Iraq, Libya, North Korea, Syria, nor any other country to
4: which the United States has prohibited export.  I will not
5: download or otherwise export or re-export the Programs,
6: directly or indirectly, to the above mentioned countries nor
7: to citizens, nationals or residents of those countries.
8:
```

[9] Sous UNIX, il s'agit de fichiers ayant l'extension .so, sous windows ce sont des dll.

[10] Sa syntaxe est la même que la variable **`$PATH`**, à savoir une liste de répertoire sans espace, séparé par deux points.

```
9: I am not listed on the United States Department of Treasury
10: lists of Specially Designated Nationals, Specially
11: Designated Terrorists, and Specially Designated Narcotic
12: Traffickers, nor am I listed on the United States Department
13: of Commerce Table of Denial Orders.

>>> textproc.grep('tmp.txt', 'citizen')
1: I am not a citizen, national or resident of, and am not
7: to citizens, nationals or residents of those countries.

>>> mygrep = textproc.grep
>>> mygrep("tmp.txt", "python")
python not found in file tmp.txt

>>> from textproc import listing
>>> listing("tmp.txt")
[...]
```

Les modules, en temps qu'objets python possèdent divers attributs qui peuvent être utiles pour récupérer dans un programme des informations sur ceux-ci. Notamment, le nom d'un module est contenu dans la variable **__name__** et la chaîne de documentation dans la variable **__doc__**. La fonction **dir** permet de lister les noms définis dans le module donné en argument[12], c'est-à-dire ses attributs et les objets qui y sont créé (fonctions, variables, classes). Un module étant un script, il peut être directement appelé en argument de l'interpréteur python. Il est alors considéré comme le module principal et reçoit comme valeur pour la variable **__name__** la chaîne `'__main__'`. Ainsi, lorsque l'on veut écrire du code pour tester un module, sans que celui-ci soit exécuté lors d'un import, on peut le placer dans le module même en le faisant précéder de l'instruction `if __name__ == '__main__'` .

Le second exemple utilise un module de la bibliothèque standard python comportant un très grand nombre d'outils :

[11] Ce texte est une partie de celui que l'on doit approuver pour télécharger le SGBD Oracle
[12] Sans argument, cette fonction liste les noms défi nis dans l'espace de nom global

```
#!/usr/bin/python -i
# programme utilisant le module sys

import sys

# affiche une aide sur ce module
print sys.__doc__

# affiche la liste des noms contenus dans sys
print dir(sys)

# sys.ps1 et sys.ps2 sont deux variables de ce module
# elle contiennent la chaîne à afficher dans le prompt python.
# on peut bien sur les modifier.
print "ps1 contient", sys.ps1
print "ps2 contient", sys.ps2
sys.ps1 = ":-) "
```

```
# sys.path est la variable contenant les répertoires
# de recherche des modules
print "répertoires : "
for directory in sys.path:
    print directory
sys.path.append("/home/toto/python")

# sys.modules contient le dictionnaire des modules chargés
print "modules :"
for mod in sys.modules.keys():
    print sys.modules[mod]

# enfin argv contient les arguments de la ligne de commande.
# vous pouvez tester ce script en exécutant :
# python script argument1 argument2 argument3
print "arguments :"
for argument in argv:
    print argument
```

4.6.3 Paquetages

Les paquetages offrent aux développeurs python le moyen de regrouper plusieurs modules dans une entité logique. Ils sont basés sur une arborescence de modules qui correspond à une arborescence des fichiers les implémentant. Ainsi, le nom du paquetage correspond à un répertoire qui peut contenir des modules et d'autres répertoires. Par exemple le paquetage **`distutils`** contient, entre autres, le module **`error`** (error.py) et le sous-répertoire **`command`** lui-même contenant le module **`build`** (build.py). On peut alors importer ces modules par : `import distutils.error, distutils.command.build`

Il est aussi possible d'importer en une seule instruction tous les modules d'un paquetage en précisant son nom comme argument de la commande import. Par exemple : `import distutils` .

Chaque répertoire du paquetage doit impérativement contenir un fichier **__init__.py** (même vide), qui sert à faire des initialisations et exécuter du code si besoin est. Notamment, on peut, dans ce fichier, attribuer une valeur à la variable **__all__** qui est une liste contenant les noms des modules composant le paquetage à importer, si par exemple on ne souhaite pas qu'ils le soient tous.

4.7 Espaces de noms

Un *espace de nom* est une relation entre des objets et des références. Un objet python existe de manière unique mais peut avoir plusieurs références (noms) dans plusieurs endroits du programme. C'est un concept important en python puisqu'il permet aussi de limiter la portée de certaines définitions telles que celles des fonctions par exemple. Ainsi deux modules peuvent avoir une fonction ayant le même nom sans que cela ne pose de problème à l'interpréteur.

4.7.1 Hiérachie

Il existe ainsi en python différents espaces de noms. On peut citer par exemple : l'espace des noms intégrés (ang. *builtins*), les noms globaux dans un module, les noms locaux dans une fonction, dans une classe ou même dans une instance de classe (un objet à proprement parlé).

Ces espaces de noms sont créés à des instants différents et ont une durée de vie différente :

- l'espace des noms intégrés est créé au démarrage de python,
- ceux des modules au moment de leur chargement,
- et ceux des classes et fonctions au moment de leur appels ; ils disparaissent lorsque l'appel est terminé.

Les espaces de noms sont gérés de manière dynamique mais la portée des variables est déterminée statiquement à la *semi-compilation*. Cela signifie que l'on ne peut pas faire référence à une variable de l'espace de nom global dans une fonction si celle-ci n'existe pas encore, même si au moment de l'appel à cette fonction on est sûr qu'elle existera.

Ainsi, il existe une véritable hiérarchie des espaces de noms dans un programme python. Le niveau le plus haut est l'espace *global* qui est en fait celui des modules (**__main__** en particulier, cf. section 4.6 page 30). Une fonction définie dans un module possède son propre espace de nom, l'espace local, et a accès aux variables de l'espace global en lecture seulement (à moins de les avoir déclarées globales par l'instruction du même nom, cf. section 4.3 page 20). De même, une classe possède un espace de nom local dans lequel sont définis les attributs de la classe. Chaque instance d'une classe (objet) a accès aux variables globales et aux variables de classes en lecture uniquement, ainsi qu'à ses propres variables (attributs) locales en lecture-écriture. Les modules possèdent aussi chacun leur espace de nom. Lorsque l'on utilise la forme `from import` de l'importation de module, les espaces de noms global et local sont alors réunis, de sorte que les variables du module importé apparaissent dans l'espace de nom du module **__main__**.

On peut donc répertorier les différents espaces de noms dans la hiérarchie suivante, chaque objet ayant accès aux espaces des objets qui le contient :

module (en particulier **__main__**)
  fonction (`def`)

classe (`class`)
méthode (`def`)
instance (**`objet = classe()`**)

Lorsqu'une variable est référencée dans une fonction mais qu'elle n'y a pas été définie, python la recherche dans l'espace de nom global. Enfin, l'instruction `del` permet de supprimer une référence dans l'espace de nom dans lequel elle a été exécutée.

 `del` ne supprime donc pas réellement l'objet référencé. En fait, elle décrémente le compteur de référence de l'objet et supprime son nom du dictionnaire de nom courant. Son espace mémoire n'est libéré que lorsqu'il n'y a plus aucune référence sur lui. Dans ce cas, le *garbage collector* se charge de le supprimer effectivement.

4.7.2 Exemple

Par exemple :

```
>>> class Espaces:
...     var = 10
...     def affiche(self):
...         print var, Espaces.var, self.var
...
>>> var = 20
>>> obj = Espaces()
>>> obj.var = 30
>>> obj.affiche()
20 10 30
```

Ici, nous avons créé une classe **`Espaces`** contenant un attribut **`var`** (valant 10) et une méthode **`affiche`**. Puis, nous avons défini une variable **`var`** (valant 20) dans l'espace de nom global. Ensuite, nous avons créé une instance de la classe **`Espaces`** de nom **`obj`** et modifié la valeur de son attribut **`var`** (valant désormais 30). Enfin, nous avons appelé la méthode **`affiche`** de **`obj`**. Celle-ci imprime **`var`** à l'écran. Python la recherche donc d'abord dans l'espace de nom local de la fonction. Ne la trouvant pas il la recherche dans l'espace global et trouve qu'elle existe, donc récupère sa valeur. Puis elle imprime **`Espaces.var`** qui est une variable défini dans la classe **`Espaces`** et enfin **`self.var`** qui est la variable d'instance de l'objet **`obj`**.

 Il convient donc d'être très vigilant lorsque l'on programme en python afin de ne pas subir des masquages dans le nom des variables et objet python en général. Il est donc conseillé de toujours utilisé une notation préfixée du nom de l'objet python contenant les variables en question.

4.8 Conclusion du guide python

Nous avons vu tout au long de ce guide les bases de la programmation en Python. Nous avons appris à utiliser dans un premier temps l'interpréteur python pour nous pencher ensuite sur les spécificités du langage à proprement parlées. Ainsi, nous avons découvert les différents types de données qu'il propose

en gardant en tête qu'il est tout à fait possible d'en créer de nouveaux, notamment par les outils de programmation orientée objet qui nous sont offerts en python. Nous avons aussi vu comment définir nos propres fonctions et comment les regrouper en unités logiques dans un module ou un paquetage. Et enfin, nous avons abordé une notion importante en python, les espaces de noms.

Ces bases vous permettent donc à présent d'écrire vos propres *scripts* ou programmes python. Ceci dit, les possibilité de développement avec ce langage sont si étendues qu'il serait illusoire de croire qu'elles sont ici toutes abordées. Il est vivement recommandé de consulter une documentation plus précise si vous souhaitez vous lancer dans un projet plus spécifique. Notamment, dans le domaine scientifique, de nombreuses bibliothèques de fonctions et classes existent déjà et peuvent vous éviter d'avoir à réinventer la roue !

Enfin, nous avons vu dans l'introduction et dans la section 4.6 qu'il est possible d'utiliser d'autres langages de programmation pour étendre les possibilités de python. Ceci a l'avantage d'apporter la vitesse d'exécution d'un code compilé (contrairement à python qui est interprété) ou même d'utiliser du code déjà existant. Ce sujet est entrevu dans l'annexe D qui décrit l'interface de programmation en C pour python ainsi que deux outils permettant de simplifier ce mécanisme.

Pour conclure, nous ne pouvons que vous conseiller de consulter les références données dans l'annexe E et en particulier le site de Python : http ://www.python.org.

A Conventions

Dans le guide python, les conventions suivantes ont été adoptées :

- Les exemples apparaissent dans des boites à fond grisé. Ils sont de deux types :

  console Ils correspondent à la sortie d'un terminal. Le symbole $ représente l'invite de commandes d'un *shell* UNIX, >>> celle de l'interpréteur python.

  ```
  $ python >>> print 'Exemple' Exemple
  ```

  script Ils correspondent au contenu d'un fichier python. En rouge apparaissent les commentaires, en bleu clair les mots clés, en vert les chaînes de caractères, en noir le reste.

  ```
  # commentaire
  motclé
  'chaîne'
  variable
  ```

- Les points requérant une attention particulière sont précédés du symbole  .
- Les mots nécessitant une définition sont écrits en *italique* et apparaissent dans le glossaire à la fin du guide.

B Codes du projet CALVI

B.1 Codes conservatifs

	VADOR	**FBM**
Contact	F. Filbet	G. Manfredi A. Ghizzo pour le 2d
Langage	C++	f77 et f90
Plate-forme	Sun, Silicon Graphics, Compaq	SX5 (vectoriel), Alpha
Parallèle	oui (MPI)	
Dimensions	1d, 2d, 1d1/2	1d, 2d, 1d2/2
Conditions aux limites	ouvertes ou périodiques	ouvertes ou périodiques
Electro(stat)(mag)	(oui) (pour le 1d)	les deux
Multi-espèces	une espèce avec fond	une ou deux espèces
Divers	basé sur une méthode dite PFC (Positive Flux Conservative)	2d est vectorisé (SX5)

B.2 Codes semi-lagrangiens

	Code1	**Code2**
Contact	A. Ghizzo	E. Sonnendrücker
Langage	f77 et f90	f90 + python + C (généré automatiquement)
Plate-forme	SX5, Origin, Power4, Mac	Sun, Silicon Graphics, Cray T3E
Parallèle	oui (Open MP, MPI)	oui (Open MP, MPI)
Dimensions	1d, 1d1/2, 1d2/2, 2d, 2d1/2	1d, 2d, 3d
Conditions aux limites	ouvertes ou périodiques	
Electro(stat)(mag)	les deux	électrostatique
Multi-espèces	une ou deux espèces	une espèce
Divers		Splitting x/v

B.3 Codes semi-lagrangiens adaptatifs

	`Moving`	`Yoda`	`Obiwan`
Contact	S. Salmon E. Sonnendrücker	M. Mehrenberger	M. Gutnic M. Haefele
Langage	f90 + python	C++	C++
Plate-forme	Sun	Sun	Sun, PC-linux
Parallèle	non	non	non
Dimensions	1d	1d	1d
Conditions aux limites	ouvertes	ouvertes ou périodiques	ouvertes ou périodiques
Electro(stat)(mag)	électrostatique	électrostatique	électrostatique
Multi-espèces	une espèce	une espèce	une espèce
Divers	maillage qui bouge	maillage adaptatif (bases hiérarchiques d'éléments finis)	maillage adaptatif (interpolettes)

C Bibliothèque standard python

Cette annexe liste les modules fournis dans la librairie standard de Python. Pour obtenir une aide plus précise sur chacun d'entre eux, vous pouvez consulter la documentation donnée en annexe E ou bien les importer dans l'interpréteur et lire la chaine `nom_module.__doc__`.

sys : accès à des paramètres et fonctions spécifiques au système ;

gc : interface au *garbage collector*, le système de libération automatique de la mémoire ;

types : noms pour les types prédéfinis ;

UserDict : classe encapsulant les objets dictionnaires ;

UserList : classe encapsulant les objets listes ;

UserString : classe encapsulant les objet chaines ;

traceback : affichage et récupération de la pile d'exécution ;

pickle : sauvegarde d'objets sous forme binaire (*serialization*) ;

string : opération courante sur les chaines ;

re : opérations avec des expressions régulières ;

pydoc : générateur de documentation et aide en ligne ;

math : fonctions mathématiques ;

cmath : fonctions mathématiques sur les nombres complexes ;

random : génération de nombres pseudo-aléatoires ;

array : matrices et tableaux ;

xreadlines : itération sur les lignes d'un fichier ;

os : diverses interfaces au système d'exploitation ;

os.path : manipulation de chemin d'accès aux fichiers ;

time : fonctions d'accès et de conversion du temps et de la date ;

socket : interface de réseau bas-niveau ;

thread : *multithreading* ;

threading : *multithreading* haut-niveau ;

D Étendre python

Cette annexe est dédiée aux possibilités d'extension de python. En effet, comme dit dans la section 4.6 page 30, il est possible d'écrire des modules python dans des langages compilés (C, C++, fortran) sous forme de bibliothèques dynamiques.

D.1 API Python/C

L'interpréteur python est écrit en langage C. Son interface de programmation permet de lui ajouter des fonctionnalités sous forme de module. Elle est directement accessible dans un programme C ou C++.

La structure classique d'un module python écrit en langage C est la suivante :

en-tête : une directive de préprocesseur `include` pour importer les définitions du fichier `Python.h` ;

fonctions : la déclaration et définition de chaque fonction du module qui prennent en paramètre deux pointeurs sur des `PyObject` et en renvoient aussi un ou renvoient `NULL` pour une erreur ;

table : la déclaration d'un tableau qui fait correspondre un nom (une chaîne), qui sera utilisé par l'interpréteur python, avec un pointeur de fonction (déclaré précédemment), et se terminant par une entrée `NULL` ;

initialisation : une fonction d'initialisation du module portant le nom `initX` où `X` est le nom de ce module.

Toutes les définitions de fonctions, types et macros sont contenues dans le fichier d'entête `Python.h`[13], dans un sous-répertoire `/usr/include/` par défaut (p.ex. `/usr/include/python2.2` pour la version 2.2). Leur nom est généralement préfixé par `Py`.

La plupart des fonctions prennent en argument le type `PyObject *` qui est un pointeur sur une structure de donnée représentant tout objet python. Pour les types de base de python, il existe une macro pour tester l'appartenance d'un objet à ce type (p.ex. `PyListCheck(a)` est vrai si l'objet <a> est une liste). Pour faire la correspondance entre objets python et type C, il existe aussi des macros utiles :

PyArg_Parse : prend en argument un objet python, une chaîne de format et la liste des adresses de variables qui récupèrent la(les) valeur(s) de l'objet. La chaîne de format permet de préciser quel objet python on attend de récupérer, par exemple `"(s)"` signifie que l'objet python est un tuple contenant un seul élément chaîne (ou *string*). Les formats sont nombreux comme `i` pour une variable entière, `L` pour un entier long, `f` pour un réel en virgule flottante, `O` pour un objet ...

Py_BuildValue : prend en argument une chaîne de format du même type que précédemment décrit, suivit des valeurs des objets correspondant au format et renvoie un pointeur sur l'objet python ainsi construit.

Il y en a de nombreuses autres, pour de plus amples détails, il est recommandé de lire la documentation sur l'API Python/C et la création de module sur le site officiel de python (cf. annexe E page 50).

Voici, par exemple, le code C d'un module `hello`[14] comportant une fonction `message` renvoyant une chaîne :

[13] Ce fichier doit être inclus avant tout autre .h, du fait de certaines définitions de macros

[14] extrait de Programming Python, 2nd edition (cf. annexe E)

```c
/*****************************************************************
 * Un module simple ecrit en C, s'appelant "hello" ; compilez-le
 * en ".so" dans PYTHONPATH, importez-le et appelez hello.message
 *****************************************************************/

#include <Python.h>
#include <string.h>


/* fonction du module :
 *     renvoie un objet
 *     prend en paramètre self (inutilisé)
 *        et args (argument passé par python lors de l'appel)
 */
static PyObject * message(PyObject *self, PyObject *args)
{
  char *fromPython, result[64];
  /* converti de Python vers C */
  if (! PyArg_Parse(args, "(s)", &fromPython))
      return NULL;  /* NULL=levée d'exception */
  else {
      /* construit une chaîne C */
      strcpy(result, "Hello, ");
      /* ajoute la chaîne Python passée en argument */
      strcat(result, fromPython);
      /* converti de C à Python */
      return Py_BuildValue("s", result);
  }
}

/* table d'enregistrement  */
static struct PyMethodDef hello_methods[] = {
  {"message", message, 1}, /* nom et pointeur de la fonction */
  {NULL, NULL}             /* fin de table */
};

/* fonction d'initialisation appelée à l'importation */
void inithello(  )
{
  /* nom du module et tableau des fonctions */
  (void) Py_InitModule("hello", hello_methods);
}
```

Pour compiler ce module, il suffit d'exécuter dans une console :

```
$ gcc hello.c -I/usr/include/python2.2 -fpic -shared -o hello.so
```

Alors, on peut tester ce module :

```
>>> import hello
>>> hello.message('world')
'Hello, world'
>>> hello.message('extending')
'Hello, extending'
```

D.2 Outils d'interfaçage

Comme vous avez pu le constatez, pour un petit module, c'est assez simple de construire l'interface qui permettra à l'interpréteur de l'importer et l'utiliser. Ceci dit, pour des projets un peu plus conséquent, cela devient très difficile et on peut avoir recours à des outils spécialisés dans ce genre de tâches. On les appels des générateurs d'interfaces ou *wrappers*. Ils génèrent ainsi, à partir d'un programme déjà existant, des fichiers à compiler avec celui-ci pour obtenir un module python fonctionnel comprenant les fonctions déclarées dans une interface.

D.2.1 swig

SWIG (Simplified Wrapper and Interface Generator) utilise des déclarations de types et fonctions en langage C/C++ pour générer un module complet d'extension C pour Python[15]. Il gère donc toutes les conversions de type, les exceptions, les compteurs de références, ...

Voyons ceci sur un exemple, similaire au précédent. Prenons le programme C suivant **`hello.c`** :

```c
#include <string.h>
#include "hello.h"

static char result[64];

char * message(char * label)
{
  strcpy(result, "Hello, ");
  strcat(result, label);
  return result;
}
```

avec le fichier entête **`hello.h`** :

```c
extern char * message(char * label);
```

Nous écrivons un fichier interface pour SWIG, **`hello.i`** :

[15] En fait, SWIG est capable de faire ce travail pour divers langage de script tel perl, tcl/tk ... (cf. http ://www.swig.org)

```
%module hello

%{
#include "hello.h"
%}

extern char * message(char * label);
```

Ce fichier contient des directives pour SWIG commençant par %. Ici, %module précise le nom du module à générer, et le code C entre les balises %{ et %} est directement recopié dans le fichier C créé. SWIG est capable de scanner un fichier .h en entier, ce qui évite encore de préciser, comme nous l'avons fait ici, les fonctions du module, en spécifiant par exemple : %include hello.h

Ensuite, il suffit d'exécuter[16] :

```
$ swig -python hello.i
Generating wrappers for Python
$ gcc -c hello.c
$ gcc -c hello_wrap.c
$ gcc hello_wrap.o hello.o -fpic -shared -o hello.so
```

pour obtenir le même module que précédemment. Ceci nous a donc évité de créer l'interface manuellement.

Ce logiciel permet aussi d'encapsuler des classes C++ dans des classes python, qu'on nomme *shadow classes*. Ainsi, l'interface de la classe python associée est strictement identique à celle de la classe C++.

Voyons maintenant un exemple[17] basé sur un programme en C++. On cherche ici à pouvoir utiliser en python une classe C++ permettant de résoudre une équation de diffusion thermique à 2 dimensions :

[16] Il est bien sûr plus judicieux de regrouper ces commandes dans un fichier Makefile.

[17] extrait modifié de la documentation de swig

```
// Fichier : pde.h
#include <math.h>
#include <stdio.h>

// classe modélisant une grille (2D)
class Grid2d {
public:
  Grid2d(int ni, int nj);
  ~Grid2d();
  double **data;
  int      xpoints;
  int      ypoints;
};

// classe pour la résolution de l'équation
class Heat2d {
private:
  Grid2d    *work;                  // grille temporaire
  double    h,k;                    // espacement de la grille
public:
  Heat2d(int ni, int nj);
  ~Heat2d();
  Grid2d    *grid;                  // données
  double     dt;                    // pas de temps
  double     time;                  // temps écoulé
  void       solve(int nsteps);     // résolution sur nsteps
  void       dump(char * filename); // sauvegarde dans un fichier
  void       set_temp(double temp); // fixe la température
};
```

Nous allons donc écrire le fichier d'interface pour swig et utiliser des directives pour :
– créer une nouvelle classe Grid2dRow représentant une ligne ;
– ajouter une méthode à Grid2d pour opérer sur une ligne.

```
%module pde
%{
#include "pde.h"
%}
%include pde.h

%inline %{
  // définition de la classe Grid2dRow
  class Grid2dRow {
    public:
      Grid2d *g;       // grille
      int    row;      // numéro de la ligne
      double __getitem__(int i) {
        return g->data[row][i];
      };
      void __setitem__(int i, double val) {
        g->data[row][i] = val;
      };
  };
%}

// ajout des méthodes get, set
// et __getitem__ retournant une ligne
%addmethods Grid2d {
  Grid2dRow __getitem__(int i) {
    Grid2dRow r;
    r.g = self;
    r.row = i;
    return r;
  };
};
```

Vous remarquerez que le nom des méthodes de la classe Grid2dRow est précédé et suivi par deux caractères souligné comme les noms spéciaux de python. Effectivement, il s'agit de méthodes spéciales, elles sont appelées par python lors de l'accès à des types de donnée comme les tuples ou les listes en lecture (**`__getitem__`**) ou en écriture (**`__setitem__`**). Cela permet de simplifier le code python correspondant puisqu'alors, les éléments d'une ligne pourront être accédés directement en donnant l'indice entre crochets. Sinon, il aurait fallu écrire des accesseurs get et set prenant en argument l'indice et la valeur. Ainsi, les lignes de code suivantes sont équivalentes :

```
L = [35.6, 54.2, 69.0]

print L[1] # équivalent à
print L.__getitem__(1)

L[2] = 26.3 # équivalent à
L.__setitem__(2, 26.3)
```

La directive **`%inline`** permet comme son équivalent C++ d'écrire du code directement après, encadré par **`%{`** et **`%}`**. Comme son nom l'indique, **`%addmethods`** permet d'ajouter une méthode à une classe.

Aprés exécution de swig et compilation du module, on peut alors utiliser ces classes comme ceci par exemple :

```
# Exemple d'utilisation

from pde import *

# conditions initiales
def initcond(h):
        h.set_temp(0.0) # méthode de la classe Grid2d
        nx = h.grid.xpoints # attribut de la classe Grid2d
        for i in range(0,nx):
                h.grid[i][0] = 1.0 # __setitem__

# initialisation et résolution d'un problème
h = Heat2d(50,50) # solver
initcond(h)
fileno = 1

for i in range(0,25):
        h.solve(100) # méthode de Heat2d
        h.dump("Dat"+str(fileno))
        print "time = ", h.time
        fileno = fileno+1

# calcul de la température moyenne sur la grille
sum = 0.0
for i in range(0,h.grid.xpoints):
        for j in range(0,h.grid.ypoints):
                sum = sum + h.grid[i][j] # __getitem__

avg = sum/(h.grid.xpoints*h.grid.ypoints)

print "Température moyenne = ",avg
```

D.2.2 f2py

Un autre générateur est f2py. Il permet de créer un module C d'extension pour Python à partir d'un programme écrit en fortran. Il prend donc en entrée un fichier fortran et crée le module .so pour python. Ce fichier peut contenir des commentaires spéciaux facultatifs réservés à f2py pour préciser certains points (comme les directives pour swig). Ils commencent par Cf2py et se terminent à la fin de la ligne.

Si le code fortran ne peut être modifié, la manière la plus simple et la plus rapide est d'appeler f2py de la manière suivante :

```
$ f2py -c fichier.f -m nom_module
```

On obtient ainsi le module nom_module que l'on peut importer dans l'interpréteur python.

A titre d'exemple, nous allons créer un module python à partir d'un code fortran effectuant le produit de deux matrices. La procédure se nomme mult et prend un certain nombre d'argument, mais grâce à

l'analyse de f2py, la fonctions appelée depuis python pourra prendre moins de paramètres. En effet, il est clair dans l'exemple suivant que A et B sont des tableaux et que m, n, o, p en sont les dimensions, elles sont donc facultatives dans le code python.

```
      SUBROUTINE MULT (A, m, n, B, o, p, C)
      INTEGER m, n, o, p, i, j, k
      INTEGER A(m,n), B(o,p), C(m,p)
cf2py intent(in) m
cf2py intent(in) n
cf2py intent(in) o
cf2py intent(in) p
cf2py intent(in) A
cf2py intent(in) B
cf2py intent(out) C
cf2py depend(m) C
cf2py depend(p) C
      INTEGER t
      DO 320 i=1, m
         DO 310 j=1, p
            t = 0
            DO 300 k=1, n
               t = t + A(i, k) * B(k, j)
 300        CONTINUE
            C(i, j) = t
 310     CONTINUE
 320  CONTINUE
      END
```

Les commentaires fortran précisent à f2py la nature des arguments avec **`intent`** (in pour les arguments en entrée et out pour les valeurs de retour). Avec **`depend`**, il est possible de spécifier qu'une variable ou sa valeur dépend d'une autre (ici, m et p sont les dimensions de C donc C en dépend).

Nous exécutons alors f2py sur ce fichier : `f2py -m ftest -c mult.f` et pouvons alors l'utiliser comme suit :

```
#!/usr/bin/python
import ftest
from Numeric import array # module Numeric

a = array([[1,2,5],[3,4,6],[7,7,7]])
b = array([[3,5,1],[8,4,9],[3,2,2]])

c = ftest.mult(a,b)

print "c = ", c
print ftest.mult.__doc__ # f2py ajoute une documentation
```

Le module Numeric est obligatoire pour travailler avec les matrices car c'est ce qu'utilise f2py pour transcrire un tableau fortran en donnée python.

Ce script donne la sortie suivante :

```
c =[[34 23 29]
 [59 43 51]
 [98 77 84]]

mult - Function signature:
  c = mult(a,b,[m,n,o,p])
Required arguments:
  a : input rank-2 array('i') with bounds (m,n)
  b : input rank-2 array('i') with bounds (o,p)
Optional arguments:
  m := shape(a,0) input int
  n := shape(a,1) input int
  o := shape(b,0) input int
  p := shape(b,1) input int
Return objects:
  c : rank-2 array('i') with bounds (m,p)
```

E Ressources

Dans cette annexe, vous trouverez une liste non exhaustive de livres et sites internet en relation avec python.

E.1 Livres

- Guido VAN ROSSUM and Fred L. DRAKE, Jr., The Python Tutorial – An Introduction to Python, Network Theory Limited, 2003
- Mark LUTZ & David ASCHER, Introduction à Python, O'Reilly, 2000
- Mark LUTZ, Programming Python, 2nd edition, O'Reilly, 2001
- Gérard SWINNEN, Apprendre à programmer avec Python, 2003
- Mark PILGRIM, Plongez au coeur de Python, 2001
- David M. BEAZLEY, Python Essential Reference, Second Edition, New Riders Publishing, 2001
- Fredrik LUNDH, Python Standard Library, O'Reilly, 2001

E.2 Sites internet

- **`www.python.org`**, site officiel du langage Python
- **`comp.lang.python`**, *newsgroup* dédié à python
- **`wikipython.flibuste.net`**, site de documentation sur python en français
- **`www.pfdubois.com/numpy/`**, site de *Numerical Python*, extension pour le calcul matriciel
- **`starship.python.net/crew/hinsen/`**, Python for science, livre en ligne
- **`www.swig.org`**, site officiel de swig, outil d'interface multilangage
- **`cens.ioc.ee/projects/f2py2e/`**, site officiel de f2py, outils d'interfacage python-fortran

E.3 Organisations utilisant python

- Google – Beaucoup de composants du moteur de recherche google sont écrits en python
- Zope Corporation – Elle a développé un puissant serveur d'applications web en python
- Los Alamos National Laboratory (LANL) Theoretical Physics Division – Elle utilise python comme module de control dans une application de calcul et modélisation physique, tournant sur des machines parallèles et *clusters*.
- NASA, Johnson Space Center – Il utilise python en tant que langage de script standard dans son application *Integrated Planning System*.
- IV Image Systems AB – Il utilise python dans beaucoup de projet, notemment un système de production d'images satellites pour l'institut météorologique et hydrologique de Suède (SMHI)

F Glossaire

Affectation : attribution d'une valeur à une variable. L'affectation sert aussi de déclaration de variable en python.

Appelable : caractéristique des objets fonctions et classes qui exécutent du code lorsqu'elle sont référencées dans une instruction.

Built-in names : nom des fonctions et variables faisant partie de l'interpréteur python et existant du démarrage à la fin de l'exécution.

Bytecode : langage intermédiaire généré par l'interpréteur python au moment de la semi-compilation pour accélérer l'exécution. Ce code peut être sauvegardé par python dans des fichiers .pyc (ou .pyo si les directives d'optimisation ont été utilisées) ce qui améliore les chargements ultérieurs.

Classe mère : voir Super classe.

Exception : interruption du cours normal d'un programme qui se produit lorsqu'une erreur ou une condition particulière est détectée durant l'exécution. Une exception transfert le control du code qui l'a provoqué à une autre partie du code, généralement une routine appelée *exception handler* ou bien au bloc englobant.

Garbage Collector : processus léger chargé de libérer la mémoire consommée par des objets qui ne sont plus référencés.

Hiérarchie de classes : organisation sous forme arborescente des liens d'héritage entre classes.

Module : ensemble de déclarations et d'instructions python regroupées dans une entité logique possédant son propre espace de nom. Un module peut être sous la forme d'un script ou d'un fichier objet .so.

Multithreading : répartition d'un programme en tâches concurrentes exécutées "simultanément". C'est une forme de multiprocessing.

Newsgroup : forums de discussion ou collection de messages (ou articles) déposés par des utilisateurs sur des serveur de news.

Par valeur : mode de passage des paramètres à une fonction. En python, ils sont toujours transmis par valeur mais cette valeur est une référence sur un objet en mémoire.

Prompt : symbole ou message apparaissant à l'écran pour indiquer qu'un programme est prêt à recevoir une commande.

Quotes : guillemets.

Script : fichier au format ASCII contenant des déclarations et des instructions python. Il peut être passé en argument à l'interpréteur ou bien directement exécuté dans un shell (auquel cas la première ligne est un commentaire spécial).

Semi-compilation : action de générer du code intermédiaire (ou bytecode) non exécutable mais interprétable par python.

Serialization : sauvegarde d'objets python sous forme binaire dans un fichier sur disque.

Shadow Class : classe générée par swig ayant pour but de fournir une interface à la classe python scrictement identique à la classe C++ d'origine. Les méthodes de cette classe ne font qu'appeler les méthodes de l'objet C++. voir wrapper.

Slice : tranche. En python, on désigne par slice une expression qui, placée entre crochets, permet de récupérer une sous-partie d'objet comme les chaines, les listes et les tuples. (p.ex. `l[2:5]`)

Super classe : classe parente de la classe concernée. Elle se trouve à un niveau supérieur dans la hiérarchie des classes.

Wrapper : objet encapsulant un autre objet pour modifier son interface et parfois son comportement.

Index

Références

[1] Nicolas Besse and Eric Sonnendrücker. Semi-lagrangian schemes for the vlasov equation on an unstructured mesh of phase space. *J. Comput. Phys.*, 191(2) :341–376, 2003.

[2] F. Filbet and E. Sonnendrücker. Comparison of Eulerian Vlasov solvers. *Comput. Phys. Comm.*, 150(3) :247–266, 2003.

[3] F. Filbet and E. Sonnendrücker. Numerical methods for the vlasov equation. In F. Brezzi, A. Buffa, S. Corsaro, and Murli A., editors, *in Numerical Mathematics and Advanced Applications, ENUMATH 2001*. Springer-Verlag, 2003.

[4] Michael Gutnic, Ioana Paun, and Eric Sonnendrucker. Vlasov simulations on an adaptive phase-space grid. Technical Report 03032, Institut de Recherche Mathématiques Avancée, Université Louis Pasteur, Strasbourg, 2003.

[5] F. Huot, A. Ghizzo, P. Bertrand, E. Sonnendrücker, and O. Coulaud. Instability of the time splitting scheme for the one-dimensional and relativistic Vlasov-Maxwell system. *J. Comput. Phys.*, 185(2) :512–531, 2003.

[6] Eric Sonnendrucker, Francis Filbet, Alex Friedman, Edouard Oudet, and Jean-Luc Vay. Vlasov simulations of beams with a moving grid. Technical Report 03031, Institut de Recherche Mathématiques Avancée, Université Louis Pasteur, Strasbourg, 2003.

Table des matières

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803